REMISE A NIVEAU EN OPENGL MODERNE

Malek.bengougam@gmail.com

L'objectif de ce document n'est pas de réexpliquer l'ensemble des concepts vu l'an passé mais de permettre une transition plus rapide vers le cours de cette année.
Je vous renvoie aux ppt pour le détail des fonctionnalités ainsi qu'à votre moteur de recherche, mais n'hésitez pas à me poser des questions si nécessaire.

Ce cours suppose l'utilisation de la classe EsgiShader qui encapsule les fonctionnalités de chargement, compilation et linkage des shaders dans un shader program.

Plutôt que de réécrire nos propres fonctions mathématiques nous allons aussi utiliser la librairie glm (qui n'est pas ma préférée mais est très répandue dans les tutoriaux sur le net).
Prendre la dernière version ici : http://glm.g-truc.net/0.9.7/index.html
Cette librairie n'est composée que de fichiers entête .hpp il n'y a donc aucun fichier .lib avec lequel il faudrait linker l'exécutable.
On aura essentiellement besoin des includes suivants (cf. le code sample sur la page web)

```
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::rotate, glm::scale, glm::perspective
#include <glm/gtc/type_ptr.hpp> // glm::value_ptr
```

On continue également à utiliser glew afin de pouvoir facilement charger les extensions qui nous intéressent (ceci n'est pas utile sous Mac OS X car elles sont déjà exposées par défaut).

A. Fonctionnalités de base

Dans le cadre de ce document nous allons nous limiter à trois fonctions que sont l'initialisation, la terminaison et le rendu. Pour un bon fonctionnement de l'application il faut également une fonction de rappel lorsque la fenêtre est redimensionnée, Resize, ainsi qu'un fonction de mise à jour de la simulation, Update. Voici un premier exemple d'utilisation d'EsgiShader :

```
EsgiShader g_BasicShader;

bool Initialise()
{
	g_BasicShader.LoadVertexShader("basic.vs");
	g_BasicShader.LoadFragmentShader("basic.fs");
	g_BasicShader.Create()

	// cette fonction est spécifique à Windows et permet d'activer (1) ou non (0)
	// la synchronisation verticale
	#ifdef WIN32
	wglSwapIntervalEXT(1);
	#endif
	return true;
}

void Terminate() {
	g_BasicShader.Destroy();
}
```

Par défaut nous utiliserons toujours la version minimale du shader (ici OpenGL 3.2, numéro de version 150) sauf lorsque cela s'avère nécessaire.

Basic.vs

```
#version 150

// 'in' remplace 'attribute' tandis que 'layout(location=0)' permet de se
// dispenser de l'usage des fonctions glBindAttribLocation() et getAttribLocation().
layout(location=0) in vec4 a_position;

void main(void) {
	gl_Position = a_position;
}
```

Basic.fs

```
#version 150

// de la meme manière, 'out' remplace 'varying' pour les variables en sortie.
out vec4 Fragment;

void main(void) {
    // auparavant en OpenGL(ES) 2 on ne pouvait specifier que gl_FragColor
    // ou gl_FragData[]. On peut maintenant renommer la variable en sortie
    Fragment = vec4(1.0);
}
```

Note : on peut aussi spécifier un layout(location=0) en sortie du FS. Cela sert dans le cas du MRT (Multiple Render Target) qui permet de dessiner dans plusieurs buffers en une passe.

I. Rendus basiques

1. afficher un triangle

Le repère de base du GPU est normalisé [-1 ;+1] pour tous les axes. On parle de coordonnées normalisées du périphérique (Normalized Device Coordinates – **NDC**).
Il s'agit d'un repère main droite ce qui signifie que le Z position pointe hors de l'écran.

```
void Resize(GLint width, GLint height) {
	glViewport(0, 0, width, height);
}

void Update() {
	// demande au systeme de fenetrage de redessiner le contenu
	glutPostRedisplay();
}

void Render()
{
	glClearColor(0.5f, 0.5f, 0.5f, 1.f);
	glClear(GL_COLOR_BUFFER_BIT);
	// alternativement on peut utiliser la nouvelle fonction glClearBufferfv()

	auto basicProgram = g_BasicShader.GetProgram();
	glUseProgram(basicProgram);

	static const float triangle[] = {
		-0.5f, -0.5f,
		0.5f, -0.5f,
		0.0f, 0.5f
	};

	// zero correspond ici a la valeur de layout(location=0) dans le shader basic.vs
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float)*2, triangle);

	glDrawArrays(GL_TRIANGLES, 0, 3);

	glutSwapBuffers();
}
```

2. triangle dans le monde

Afin d'effectuer le même rendu mais en simulant un monde plus large il nous faut une matrice de projection (orthographique ou perspective) ainsi qu'une matrice pour simuler la caméra, et bien entendu une matrice pour placer et orienter l'objet dans le monde.
La matrice de projection a pour rôle de transformer les coordonnées exprimées dans le repère du monde (le plus souvent relativement au point de vue de la caméra) vers les coordonnées normalisées.

Ces matrices sont communes pour l'ensemble des vertices de l'objet, on les passera donc en tant que variable uniforme. Les variables uniformes ont aussi un « location » mais il n'est pas possible d'utiliser layout(location) pour spécifier cet identifiant.
On est donc obligé de continuer à utiliser la fonction **glGetUniformLocation**() pour récupérer cette information que l'on devra passer aux fonctions **glUniform*****() qui permettent de mettre à jour les données côté GPU.

Basic.vs

```
#version 150

layout(location=0) in vec4 a_position;

uniform mat4 u_worldMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_projectionMatrix;

void main(void)
{
	gl_Position = u_projectionMatrix * u_viewMatrix * u_worldMatrix * a_position;
}
```

Côté C++ on va stocker nos matrices projection et vue dans une structure ViewProj et la matrice monde dans une structure Object.

```
struct ViewProj
{
	glm::mat4 viewMatrix;
	glm::mat4 projectionMatrix;
} g_Camera;

struct Object
{
	glm::mat4 worldMatrix;
} g_Objet;
```

La fonction **glUniformMatrix4fv**() nécessite l'adresse mémoire de la matrice. Pour obtenir l'adresse d'un type glm il faut utiliser la fonction **glm::value_ptr**()

```
// inserez ces lignes dans Render() apres glUseProgram(basicProgram);

// [...]
g_Camera.projectionMatrix = glm::perspectiveFov(45.f, (float)width, (float)height,
0.1f, 1000.f);
glm::vec4 position = glm::vec4(0.0f, 0.0f, 2.0f, 1.0f);
g_Camera.viewMatrix = glm::lookAt(glm::vec3(position), glm::vec3(0.f), glm::vec3(0.f,
1.f, 0.f));

auto projLocation = glGetUniformLocation(basicProgram, "u_perspectiveMatrix");
glUniformMatrix4fv(projLocation, 1, GL_FALSE,
glm::value_ptr(g_Camera.projectionMatrix));

auto viewLocation = glGetUniformLocation(basicProgram, "u_viewMatrix");
glUniformMatrix4fv(viewLocation, 1, GL_FALSE, glm::value_ptr(g_Camera.viewMatrix));

auto worldLocation = glGetUniformLocation(basicProgram, "u_worldMatrix");
glUniformMatrix4fv(worldLocation, 1, GL_FALSE, glm::value_ptr(g_Objet.worldMatrix));
// [...]
```

Exercice 1 (obligatoire, peut-être fait en même temps que l'exercice 2)

Affichez une dizaine de triangles à l'écran avec une position et une couleur différente.
La couleur devra être passée au Fragment Shader en tant que variable uniforme.

B. Fonctionnalités avancées

Le principal problème de l'exemple précédent est d'une part qu'on est obligé de retransmettre les informations topologiques au GPU à chaque trame. Ceci n'est pas bien grave pour un simple triangle mais pour des modèles 3D à plusieurs milliers de triangle cela commence à faire beaucoup en termes de bande passante.

De plus au fur et à mesure que l'on va complexifier nos attributs (ajouts de coordonnées de texture, normales, information de skinning etc…) et que l'on va avoir des objets qui utilisent des attributs différents le driver va effectuer un travail redondant de reconfiguration des attributs en entrée du Vertex Shader.

OpenGL permet le stockage de données en mémoire via le mécanisme des Buffer Objects.
Un BO se créé à l'aide de la fonction **glGenBuffers**() qui permet de réserver un identifiant que l'on peut détruite avec la fonction **glDeleteBuffers**() qui libère également la mémoire allouée.
Cette mémoire est allouée par un appel à la fonction **glBufferData**().
En OpenGL on ne peut travailler que sur un seul buffer à la fois, il faut donc spécifier quel est le buffer qui doit être référencé à l'aide de la fonction **glBindBuffer**().

Le but final est d'essayer d'avoir le moins de code de gestion dans la boucle de rendu et d'essayer de tout faire à l'initialisation.

II. Vertex Buffer Objects (VBO)

Le premier objectif est d'éviter la redéfinition à chaque trame des données topologiques et, lorsque l'on sait que les données ne vont pas être modifiées, qu'elles restent résidentes en mémoire vidéo. On dispose d'un GLenum qui permet de spécifier à OpenGL comment seront utilisées ces données.

Le principal changement au niveau du code va être de couper-coller la partie définissant le triangle et de la mettre dans la fonction d'initialisation. On ajoute aussi le code de création du VBO variante d'un BO qui sert à stocker les attributs de vertex.

a. Ajouter une variable GLuint VBO dans la structure Object.

b. Code à ajouter dans Initialise()

```
// [...]
	static const float triangle[] = {
		-0.5f, -0.5f,
		0.5f, -0.5f,
		0.0f, 0.5f
	};

	glGenBuffers(1, &g_Objet.VBO);
	glBindBuffer(GL_ARRAY_BUFFER, g_Cube.VBO);
	// glBufferData alloue et transfort 4 * 3 octets issus du tableau triangle[]
	glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 3, triangle, GL_STATIC_DRAW);
// [...]
	glBindBuffer(GL_ARRAY_BUFFER, 0);
// [...]
```

c. Code de Terminate()

```
void Terminate() {
	g_BasicShader.Destroy();
	glDeleteBuffers(1, &g_Object.VBO);
}
```

d. Code de la partie de dessin dans Render

```
// [...]

	glBindBuffer(GL_ARRAY_BUFFER, g_Object.VBO);
	glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float)*2, 0);
	glEnableVertexAttribArray(0);

	glDrawArrays(GL_TRIANGLES, 0, 3);

// [...]
```

Notez que le dernier paramètre de **glVertexAttribPointer**() est devenu **0** (ou **NULL**, **nullptr**). Ceci est obligatoire car glVertexAttribPointer teste la valeur de GL_ARRAY_BUFFER. Lorsque celle-ci est différente de 0 (lorsqu'un VBO est bind) la fonction interprète le dernier paramètre non plus comme une adresse absolue mais un offset (ou adresse relative).

Il existe d'autre type de BO que nous allons voir au fur et à mesure. On utilisera en particulier les IBO, pour Index Buffer - aussi appelés EBO, avec E pour Element - qui servent à stocker la topologie, c'est-à-dire comment doit-on relier les vertices entre elles.

Exercice 2 (recommandé).

Modifiez le code précédent afin de dessiner le triangle en utilisant un Index Buffer. Les étapes sont les mêmes que pour un VBO à cela près que le type est **GL_ELEMENT_ARRAY_BUFFER** et qu'il faut dessiner à l'aide de la fonction **glDrawElements**() à la place de **glDrawArrays**().

III. Vertex Array Objects (VAO)

L'autre problématique réside dans le fait que l'on est obligé de re-spécifier à chaque trame le format des attributs en entrée du Vertex Shader alors que la correspondance (mapping) est constante pour le rendu d'un objet avec un shader.
Un VAO est un objet OpenGL qui permet de stocker les états des attributs (le nom Vertex Array peut prêter à confusion, mais il s'agit bien ici des paramètres et non du contenu des attributs).

Un VAO va surtout enregister le « mapping » entre un VBO et un ou plusieurs attributs et donc aussi les états liés à ces attributs (taille et type des données, tableau ou valeur simple etc…).

Note : En OpenGL l'usage d'un VAO introduit un couplage fort avec un VBO. On verra plus tard comment il est possible d'avoir un mécanisme plus souple sans ce couplage.

a. Ajouter un GLuint dans la structure Object pour stocker l'identifiant du VAO

Attention ! A partir du moment où un VAO est actif, il enregistre toutes les commandes de type glBindBuffer(), glVertexAttribPointer(), glEnable/DisableVertexAttribArray(), et d'autres

fonctions que nous verrons plus tard. Il est donc important de rétablir le VAO par défaut (0) une fois que l'on a terminé l'enregistrement.

b. Initialisation : on va donc déplacer le code de définition des attributs ici

```
// [...]
	// a ajouter apres la création du VBO

	glGenVertexArrays(1, &g_Object.VAO);
	glBindVertexArray(g_Object.VAO);
	glBindBuffer(GL_ARRAY_BUFFER, g_Object.VBO);
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float) * 3, nullptr);
	// tres important ! toujours desactiver les VAO lorsque l'on ne s'en sert plus
	glBindVertexArray(0);

// [...]
```

c. Ne pas oublier le terminate

```
void Terminate() {
	g_BasicShader.Destroy();
	glDeleteBuffers(1, &g_Object.VBO);
	glDeleteVertexArrays(1, & g_Object.VAO);
}
```

d. Le code de rendu du triangle dans la fonction render() devient alors

```
// [...]

	glBindVertexArray(g_Object.VAO);
	glDrawArrays(GL_TRIANGLES, 0, 3);

	glBindVertexArray(0);
// [...]
```

Note : Un VAO n'enregistre que ce qui est nécessaire à la définition des attributs.

Cela ne concerne donc pas les indices, et, dans le cas d'un rendu utilisant **glDrawElements**(), il sera toujours nécessaire de faire en premier lieu un glBindBuffer() de l'EBO.

IV. Uniform Buffer Objects (UBO)

On a vu comment réduire la charge sur la bande passante pour les attributs avec les VBO ainsi que le nombre de commande OpenGL avec le VAO. Cependant on se retrouve toujours à envoyer à chaque frame, qui plus est séparément, les matrices de projections et de vues alors qu'elles sont censées être constantes pour l'ensemble d'une trame.
Autrement dit, si l'on dessine plusieurs objets ces derniers vont utiliser ces mêmes matrices.
C'est pourquoi depuis OpenGL 3 on a la possibilité de partager les données uniformes entre plusieurs shaders via les Uniform Buffer Objects (UBO).

Le principe de base reste similaire à celui des VBO ou autres BO, on va là encore utiliser les fonctions glGenBuffers(), glDeleteBuffers(), glBindBuffer() et glBufferData() pour manager notre UBO.

La différence réside en pratique dans le type de BO qui devient **GL_UNIFORM_BUFFER** et dans la façon dont on va rendre accessible aux shaders nos UBO.

Premièrement il nous faut un GLuint afin de stocker l'identifiant du BO et ensuite on va simplement allouer de la mémoire pour stocker nos deux matrices (à l'initialisation).

```
glGenBuffers(1, &g_Camera.UBO);
glBindBuffer(GL_UNIFORM_BUFFER, g_Camera.UBO);
// notez le GL_STREAM_DRAW pour indiquer que les donnees changent peu par frame
glBufferData(GL_UNIFORM_BUFFER, sizeof(glm::mat4) * 2, NULL, GL_STREAM_DRAW);

glBindBufferBase(GL_UNIFORM_BUFFER, bindingPoint, g_Camera.UBO);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

La fonction glBindBufferBase() permet d'indiquer à OpenGL sur quel point d'ancrage (binding point) il sera possible de lire ces données partagées.

La variable bindingPoint contient le numéro (un entier) d'un point d'ancrage virtuel qui va nous permettre de faire le lien entre l'UBO et l'uniform block dans le shader.

Qu'est- ce qu'un uniform block ? C'est simplement une espèce de structure qui regroupe en un bloc (entre accolade) un ensemble de variables uniformes.
Dans le Vertex Shader basic.vs on va donc maintenant avoir le code suivant :

```
uniform mat4 u_worldMatrix;
layout(std140) uniform ViewProj
{
	mat4 u_viewMatrix;
	mat4 u_projectionMatrix;
};
```

`layout(std140)` n'est pas obligatoire dans notre cas et sert juste à indiquer comment les données sont alignées en mémoire (alignement de la structure, taille du bourrage etc…).

Il faut aussi spécifier a OpenGL comment lier un shader avec le point d'ancrage virtuel. Cela se fait à l'aide de ces deux fonctions. La première récupère l'équivalent du location de l'uniform block tandis que la seconde effectue concrètement le lien (toujours à l'initialisation).

```
auto blockIndex = glGetUniformBlockIndex(program, "ViewProj");
glUniformBlockBinding(program, blockIndex, blockBinding);
```

A partir de maintenant, toutes les modifications sur l'UBO seront automatiquement répercutées au niveau des shaders. Plus besoin d'updater ces variables uniformes dans chacun des shaders. Le code de mise à jour des matrices devient alors le suivant :

```
g_Camera.projectionMatrix = …
g_Camera.viewMatrix = …

glBindBuffer(GL_UNIFORM_BUFFER, g_Camera.UBO);
glBufferData(GL_UNIFORM_BUFFER, sizeof(glm::mat4) * 2,
glm::value_ptr(g_Camera.viewMatrix), GL_STREAM_DRAW);
// on peut aussi utiliser glBufferSubData, pas forcement plus optimal
//glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4) * 2,
glm::value_ptr(g_Camera.viewMatrix));
auto worldLocation = glGetUniformLocation(basicProgram, "u_worldMatrix");
glUniformMatrix4fv(worldLocation, 1, GL_FALSE, glm::value_ptr(g_Objet.worldMatrix));
```

C. OpenGL en pratique : Plus de réalisme

V. Textures et samplers

En OpenGL les textures sont représentées sous forme d'un tableau contigu. OpenGL utilise une structure spéciale nommée Texture Object, analogue à un BO, qui stocke les informations concernant la description (taille, format, données des pixels…) et son utilisation (répétition, filtrage, mipmaps…).

On crée un Texture Object avec **glGenTextures**(), on le détruit avec **glDeleteTextures**() et on le défini comme Texture Object courant avec **glBindTexture**().
Pour le moment nous allons surtout voir des textures de type 2D, donc **GL_TEXTURE_2D**.

Pour charger des images on va utiliser la librairie stb qui, en plus d'être très simple à utiliser (seulement du .h, pas de .lib à linker), supporte un grand nombre de format.
https://github.com/nothings/stb on s'intéressera surtout à l'include stb_image.h. Exemple d'utilisation :

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

bool LoadAndCreateTextureRGBA(const char *filename, GLuint &texID)
{
	glGenTextures(1, &texID);
	glBindTexture(GL_TEXTURE_2D, texID);

	// il est obligatoire de specifier une valeur pour GL_TEXTURE_MIN_FILTER
	// autrement le Texture Object est considere comme invalide
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

	int w, h;
	uint8_t *data = stbi_load(filename, &w, &h, nullptr, STBI_rgb_alpha);
	if (data) {
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);

		stbi_image_free(data);
	}
	return (data != nullptr);
}
```

Ne pas oublier de faire un **glDeleteTextures**() en fin de programme (ou de niveau) !

Note : En OpenGL 3.3+ la fonction glTexStorage2D est préférable à glTexImage2D. En effet glTexImage2D reste une fonction générique qui permet de réallouer/reconfigurer une texture ce qui est très couteux. Les textures immuables (Immutable Texture) ont des paramètres fixes et sont donc traités plus rapidement par le pilote. Exemple :

```
// Allocation d'une texture, le deuxieme parametre indique le nombre de mipmap
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
// remplissage
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

Côté shader on dispose d'un type particulier appelé « sampler » (échantillonneur) qui permet de lire le contenu de la texture en fonction de valeurs normalisées nommées coordonnées de texture. Il existe un type spécialisé par type de texture (2D, 3D, shadow, etc…).

Les coordonnées de texture sont définis dans un repère normalisé [0, 1] dont l'origine se trouve en bas à gauche. Il existe des fonctions pour modifier la position de ce repère, cf. cours.

Basic.fs (partiel)

```
uniform sampler2D u_diffuseMap;
void main(void)
{
//[…]
	vec4 diffuseColor = texture(u_diffuseMap, v_UV);
	Fragment = diffuseColor;
```

Pour que le sampler sache quelle texture doit être utilisée il faut d'une part faire un « bind » du Texture Object sur une unité de texture (par défaut ce sera toujours l'unité numéro 0). Lorsqu'on utilise qu'une seule texture à la fois, un appel à glBindTexture() est suffisant.

Cependant dans le cas où l'on doit utiliser plusieurs textures en même temps il faut définir l'unité de texture sur laquelle on va « bind » le Texture Object et informer le shader en attribuant une valeur (qui correspond au numéro de l'unité de texture) aux variables de type sampler.

Exemple avec l'utilisation de 3 textures : diffuse, masque spéculaire et normal map.

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, mesh.TexDiffuse);
auto diffuseMapLocation = glGetUniformLocation(basicProgram, "u_diffuseMap");
glUniform1i(diffuseMapLocation, 0);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, mesh.TexSpecular);
auto specularMapLocation = glGetUniformLocation(basicProgram, "u_specularMap");
glUniform1i(specularMapLocation, 1);
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, mesh.TexNormal);
auto normalMapLocation = glGetUniformLocation(basicProgram, "u_normalMap");
glUniform1i(normalMapLocation, 2);
```

Dans le fragment shader on a donc ici 3 samplers

```
uniform sampler2D u_diffuseMap;
uniform sampler2D u_specularMap;
uniform sampler2D u_normalMap;
```

Note : A partir d'OpenGL 4.2 il est aussi possible d'utiliser **layout(binding=)** pour un sampler. Ce qui supprime l'utilité de glUniform1i côté code client.

Exercice 3 (optionnel)

Faites une copie du projet précédent que vous renommerez Quad. Modifiez le code pour afficher un carré texturé. Il faut donc ajouter un vertex supplémentaire ainsi que des valeurs d'attributs supplémentaires pour les coordonnées de textures.
Les shaders doivent être adapté afin de pouvoir utiliser les coordonnées de texture et le sampler.

VI. Normales

En 3D il s'avère très vite nécessaire d'utiliser des normales principalement pour définir la façon dont une face (ou un groupe de face dans le cas des normales aux vertices) sera ombrée.

La plupart du temps ces données sont fournies directement par le logiciel de modélisation.
On va utiliser la librairie tiny_obj_loader afin de charger des fichiers de type obj.
https://github.com/syoyo/tinyobjloader
Il suffit d'ajouter les fichiers tiny_obj_loader.h et tiny_obj_loader.cc à votre projet.

```
std::vector<tinyobj::shape_t> shapes;
std::vector<tinyobj::material_t> materials;
//[...]
std::string err = tinyobj::LoadObj(shapes, materials, "cube.obj");
```

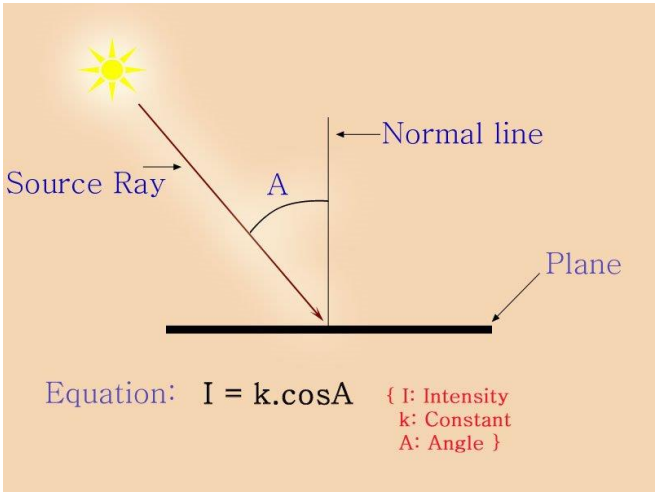
Pour chaque shape_t on peut accéder à la definition du mesh via `shapes[i].mesh`

La structure définissant la variable mesh contient les informations de position, coordonnées de texture et normales (si existantes) ainsi que la liste des indices pour la forme actuelle.

Exercice 4 (obligatoire)

Définir le projet RotatingCubes comme projet de démarrage.
Pour le moment le cube est simplement chargé en mémoire système via tiny_obj_loader.
Ajouter le code nécessaire pour que le cube s'affiche et tourne à l'écran.
Note : les shaders ont déjà été codé, il n'y a que la partie C++ à faire. Utilisez glMapBuffer() et glUnmapBuffer() pour copier les attributs dans le VBO.

Le rendu se fait actuellement à l'aide de la loi du cosinus de Lambert utilisée pour simuler un éclairage diffus provenant d'une source de lumière distante (ex : le soleil)



Exercice 5 (contrôle continu)

Créer une copie du projet RotatingCubes nommé ObjLoader afin de pouvoir charger des modèles Obj arbitraires (avec ou sans coordonnées de texture, avec ou sans normales).
Retirez les boucles for x, y, et z ainsi que u_objectColor qui ne sont pas utile pour ce projet.

D. Fonctionnalités modernes

VII. Geometry Shaders

A partir d'OpenGL 3.x un nouveau type d'unité de calcul shader à fait son apparition officielle : le Geometry Shader. Il est toutefois possible d'utiliser un Geometry Shader en OpenGL 2.x si le hardware et le pilote le permettent via l'extension Ext_Geometry_Shader4 mais la syntaxe est sensiblement différente des Geometry Shaders en OpenGL 3.+.

Tout GPU supportant la norme OpenGL >= 3.1 contient un support des Geometry Shaders. Le geometry Shader est par contre optionnel en OpenGL ES 3.1 et inexistant (sauf extension) avant.

Un Geometry Shader s'intercale entre le Vertex Shader et le Fragment Shader.
Il reçoit en entrée un tableau contenant les sorties de tous les Vertex Shaders exécutés pour le traitement d'une primitive.
Ainsi, si la primitive est GL_POINTS, le tableau ne contiendra qu'un seul vertex. Dans le cas où la primitive est GL_LINES ou GL_LINE_LIST ou GL_LINE_LOOP il en recevra deux, et trois vertices dans le cas de GL_TRIANGLES, GL_TRIANGLE_STRIP et GL_TRIANGLE_FAN.

On peut spécifier le type de données en entrée et en sortie du Geometry Shader de cette façon

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
```

Dans cet exemple on suppose donc un appel de dessin de type GL_TRIANGLES.
La métadata max_vertices permet d'indiquer le nombre de vertices que va produire en sortie notre GS. Ceci est important pour le pilote afin qu'il puisse réserver l'espace nécessaire à l'exécution du shader dans la mémoire dédiée au GS du GPU.

PRIMITIVE GL	TYPE EN ENTREE	NOMBRE DE VERTEX
GL_POINTS	points	1
GL_LINES	lines	2
GL_LINE_LIST	lines	2
GL_LINE_STRIP	lines	2
GL_TRIANGLES	triangles	3
GL_TRIANGLE_STRIP	triangles	3
GL_TRIANGLE_FAN	triangles	3

Lorsqu'il est actif le GS prend donc en entrée les sorties (out) du Vertex Shader, en premier lieu desquelles gl_Position, et émet en sortie les informations nécessaires au rasterizer qui seront interpolées et envoyées en entrée (in) du Fragment Shader.

Un Geometry Shader ne peut produire que trois types de données en sortie :
- points
- line_strip
- triangle_strip

Pour chaque vertex en sortie il faut au moins affecter une valeur à gl_Position et terminer par un appel à la fonction **EmitVertex**() qui indique la fin de la description du vertex. **EndPrimitive**() quant à elle permet optionnellement d'indiquer la fin d'une primitive.
EndPrimitive() permet dans le cas de line_strip et triangle_strip de générer des primitives disjointes.

Exemple d'un basic.gs qui ne fait qu'envoyer en sortie les données telles qu'elles ont été reçues:

```
#version 150

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main(void)
{
	// gl_in.length() retourne 3 car nous avons un triangle en entree
	for (int i = 0; i < gl_in.length(), i++) {
		gl_Position = gl_in[i].gl_Position;
		EmitVertex();
	}
	// EndPrimitive(); // optionnel ici
}
```

Si l'on remplace la ligne
`layout(triangle_strip, max_vertices = 3) out;`
par
`layout(points, max_vertices = 3) out;`

le Geometry Shader produira seulement les 3 vertices en entrée sous forme de points isolés (qui pourront d'ailleurs être grossi par l'usage de glPointSize() par ex.)

Exemple d'un Geometry Shader produisant des lignes (arêtes) à partir de triangles.
Il s'agit presque du même code à l'exception du premier vertex qui est répété pour former une succession (bande = strip) de 3 lignes à partir de 4 vertex émis : (0,1) (1,2) (2, 0 bis)

```
#version 150

layout(triangles) in;
layout(line_strip, max_vertices = 4) out;

void main(void)
{
	for (int i = 0; i < gl_in.length(), i++) {
		gl_Position = gl_in[i].gl_Position;
		EmitVertex();
	}
	gl_Position = gl_in[0].gl_Position; // 0 bis
	EmitVertex();
}
```

Un autre usage très important, mais relativement limité est l'amplification. Cela consiste à créer dynamiquement de nouvelles primitives à partir des données en entrée.

Voir https://open.gl/geometry pour un exemple d'amplification à partir d'un point.

Exercice 6 (recommandé)

Afficher une grille 2D en fil de fer calée sur le plan XZ (sol) et centrée sur l'origine à partir d'un seul appel à glDrawArrays(GL_POINTS, 0, 1) et un point positionné en (0, 0, 0).

A contrario on peut aussi décider de ne sélectionner qu'une partie des primitives reçues, par exemple en fonction l'orientation de la normale de la face. Cela revient à faire du face culling.

Voici comment effectuer le culling dans le geometry shader. Il faut modifier le Vertex Shader pour que celui-ci écrivent dans gl_Position non pas des coordonnées en NDC mais dans le repère de la caméra.

On profitera ainsi du fait que la direction de la caméra est toujours (0, 0, -1) dans le repère de vision (View Space) donc qu'un vecteur normalisé allant de la primitive à la camera aura toujours la valeur (0, 0, 1).

culling.vs (similaire à basic.vs)

```
void main(void) {
// [...]
	// gl_Position se trouve en View Space a ce moment precis
	gl_Position = u_viewMatrix * u_worldMatrix * a_position;
// [...]
}
```

culling.gs

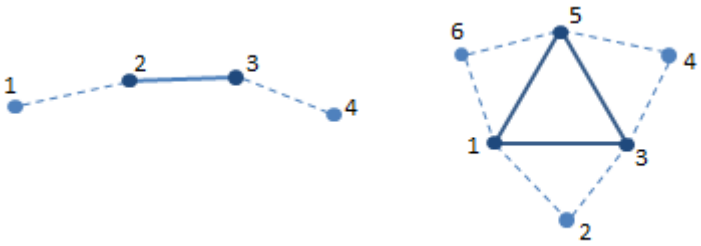
```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
// merci les UBO
uniform ViewProj
{
	mat4 u_viewMatrix;
	mat4 u_projectionMatrix;
};

// vecteur allant de l'objet a la camera
const vec3 eyeDirection = vec3(0.0, 0.0, 1.0);

void main(void)
{
	vec3 v1 = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
	vec3 v2 = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
	vec3 normal = normalize(cross(v1, v2));

	if (dot(normal, eyeDirection) >= 0.0)
	{
		for (int i = 0; i < gl_in.length(); i++) {
			// n'oubliez pas, le fragment shader veut une position en NDC
			gl_Position = u_projectionMatrix * gl_in[i].gl_Position;
			EmitVertex();
		}
	}
}
```

Parmi les multiples autres techniques sachez qu'il est possible d'utiliser le Geometry Shader pour faire du frustum culling côté GPU, de faire des extrusions pour des calculs de CSG ou pour le rendu d'ombres volumétriques (shadow volumes) ou encore pour afficher une silhouette.

Les possibilités sont multiples mais le Geometry Shader reste parfois limité par le nombre de primitives qu'il peut générer. Sur certaines plateforme l'utilisation d'un GS entraine un ralentissement du pipeline ce qui dégrade parfois les performances. Il faut donc se fier au recommandation du constructeur quant à l'usage du GS.

Données d'adjacence

Un Geometry Shader supporte les trois types de primitives usuelles, points, lignes et triangles ainsi que deux autres primitives que sont lignes avec adjacences (à gauche) et triangle avec adjacences (à droite) – les indices sont ici numérotés à partir de 1 mais pas en pratique (0).



Le type OpenGL à utiliser avec glDrawArrays() ou glDrawElements() et leurs variantes, pour ces deux nouvelles primitives est **GL_LINE_ADJACENCY** et **GL_TRIANGLE_ADJACENCY**.
Ces primitives sont composées respectivement de quatre et six vertices.

PRIMITIVE GL	TYPE EN ENTREE	NOMBRE DE VERTEX
GL_LINE_ADJACENCY	lines_adjacency	4
GL_LINE_STRIP_ADJACENCY	lines_adjacency	4
GL_TRIANGLE_ADJACENCY	triangles_adjacency	6
GL_TRIANGLE_STRIP_ADJACENCY	triangles_adjacency	6

Ces données d'adjacence sont parfois nécessaires car un Geometry Shader ne peut accéder qu'aux données de la primitive courante (point, ligne ou triangle).
Dans certains cas on aimerait avoir accéder aux primitives adjacentes. Cela nécessite donc de modifier les indices (idéalement, sans avoir à ajouter / supprimer des vertices du tableau) du maillage.
Nous verrons en cours comment calculer les indices d'adjacence à partir d'indices de triangle standard.

VIII. Interface blocks

L'ajout d'un Geometry Shader ne se fait pas sans conséquence sur le nom des variables.
Ce qui peut poser un problème lorsque l'on souhaite ré-utiliser un Vertex Shader ou un Fragment Shader dans plusieurs programmes qui ne vont pas forcément utiliser un Geometry Shader.

La solution proposée, et préférée, par OpenGL est de passer par un bloc d'interface (interface block). En gros on ne va plus transférer nos variables unes à unes d'un shader à l'autre mais en un bloc, comme une structure.

Ceci a de multiples avantages comme le fait que l'on n'est plus obligé d'avoir un nom de variable similaire dans tous les shaders, ni que l'on soit obligé de faire correspondre les locations. Il faut juste qu'il y'ait une correspondance entre les noms de bloc ainsi que les noms et les types des variables membres du bloc.

Exemple de code problématique

Problem.vs

```
out vec4 o_color;
out vec2 o_texCoords;
```

Problem.fs

```
in vec4 o_color;
in vec2 o_texCoords;
```

Problem.gs

```
// sortie du VS
in vec4 o_color[];
in vec2 o_texCoords[];

// entrée du FS, ERREUR : les variables portent le même nom en entrée et en sortie !
out vec4 o_color;
out vec2 o_texCoords;
```

L'utilisation d'un bloc d'interface permet de réduire ce problème

NoProblemo.vs

```
out VertexData {
	vec4 color;
	vec2 texCoords;
} OUT;

	OUT.color = … ;
	OUT.texCoords = … ;
```

NoProblemo.fs

```
in VertexData {
	vec4 color;
	vec2 texCoords;
} IN;

	Fragment = IN.color * texture(diffuseSampler, IN.texCoords);
```

NoProblemo.gs

```
in VertexData {
	vec4 color;
	vec2 texCoords;
} IN[];

out VertexData {
	vec4 color;
	vec2 texCoords;
} OUT;

	OUT.color = IN[i].color;
	OUT.texCoords = IN[i].texCoords;
	EmitVertex();
```

IX. Hardware Instancing

Tbd.

Exercices complémentaires :

a. Intégrez le code de camera de cet article : http://learnopengl.com/#!Getting-started/Camera . Trouver comment transformer le déplacement de la caméra pour que l'on orbite autour d'une cible en pouvant avancer et s'éloigner de la cible.

b. Afficher un quad subdivisé en 5x5 (6 vertices par cotés) rendu avec GL_TRIANGLES. Effectuer le même rendu à l'aide de GL_TRIANGLE_STRIP pour dessiner le quad subdivisé en un seul appel à glDrawArrays ou glDrawElements.

c. Ecrire un Geometry Shader qui anime les faces d'un objet en les faisant grossir et rapetisser en suivant une fonction sinusoïdale positive et d'un temps continu.
Cf.

d. Ecrire un Vertex Shader qui déplace les vertices suivant une fonction sinusoïdale en les déplaçant dans la direction de la normale.
Cf. http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter06.html